Semiannual Technical Summary

Restructurable VLSI Program

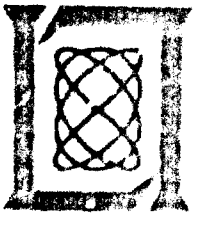
31 March 1980

Prepared for the Defense Advanced Research Projects Agency
under Electronic Systems Division Contract F19628-80-C-0002 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS



The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract F19628-80-C-0002 (ARPA Order 3797).

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Raymond L. Loiselle

Raymond L. Loiselle, Lt. Col., USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

RESTRUCTURABLE V. M PROGRAM

SEMIANNUAL TECHNICAL SUMMARY REPORT
TO THE
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

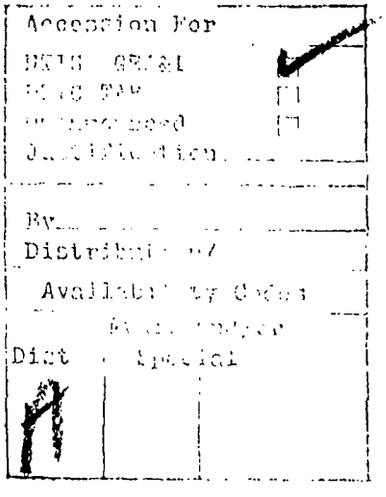
1 APRIL 1979 – 31 MARCH 1980

ISSUED 20 JANUARY 1981

LEXINGTON MASSACHUSETTS

ABSTRACT

This initial report describes work on the Restructurable VLSI Research Program sponsored by the Information Processing Techniques Office of the Defense Advanced Research Projects Agency during the two semiannual periods, covering 1 April 1979 through 31 March 1980.



CONTENTS

RESTRUCTURABLE VLSI PROGRAM

I. INTRODUCTION AND SUMMARY

The objective of the Restructurable Very Large Scale Integration (RVLSI) Program is to develop and demonstrate techniques which will make possible integration of large systems as single-package modules. We are developing techniques for restructuring large-area IC chips after fabrication in order to provide access for testing, perform defect avoidance and customization, and reconfigure a system while it is being used. The DARPA-sponsored program which is reported here is focused on development or architectural concepts for data- and signal-processing systems for RVLSI implementation, development of design aids for unique RVLSI design problems, and development of test techniques suitable for RVLSI applications. A companion Air Force-supported program in technology development and fabrication of large-area RVLSI is reported on in the Lincoln Laboratory Advanced Electronic Technology Quarterly Technical Summaries.

Since this is the first report on this program, an overview of the goals and proposed techniques is given in Sec. II.

In Sec. III the functional requirements on programmable connections or links are presented and three types of links are described and compared. In the near term the emphasis will be on laser programmed links and links made from standard logic circuitry; a longer-term goal is development of an electrically programmable nonvolatile link.

In Sec. IV we describe a hardware description language designed for efficient description of hierarchical and iterative structures of digital circuits. Since each RVLSI chip may have a different wiring configuration, complete automation of the signal routing process is essential. Solutions for this problem are presented in Sec. IV.

Results of an investigation of mapping a regular locally connected array of processing elements onto a physical array with defective elements are presented in Sec. V. An integrator for a spread-spectrum packet radio receiver has been chosen for a first implementation. The system and its partitioning into cells is described and a scheme for assignment of cells is presented.

II. PROGRAM OVERVIEW

A. INTRODUCTION

Digital LSI technology is rapidly evolving toward whole wafer systems in which the combined effects of smaller feature size and larger chip area will permit the design of monolithic structures that may be two to three orders of magnitude larger in gate count than existing LSI products. The vast amounts of computing and signal-processing power that will be possible on single substrates will allow for the introduction of more sophisticated and effective communications and sensor systems onto military platforms that are currently limited by constraints of equipment size, weight, and power consumption. In addition, a quantum jump in monolithic circuit capability has significant implications in terms of cost, reliability, and maintainability.

The increased level of integrated system complexity afforded by larger chip areas and smaller device geometries will require entirely different methods of system design, fabrication, and testing than are currently used in the integrated circuit industry. For example, a major constraint on the complexity of present day devices is the requirement that a chip be entirely free of defects in order for it to be considered useful. A more effective design and fabrication strategy in more complex systems would be to anticipate a given defect density and to provide mechanisms by which the defects, once identified, could be circumvented through redundant circuitry. Increased complexity also implies more extensive testing requirements, both to validate the functional integrity of logical modules for which there may be no direct access from the chip periphery and to identify defective cells in order that they may be selectively avoided. The lack of suitable VLSI testing and defect avoidance techniques is a major obstacle to the accelerated development of large-area high-density integrated circuit technology.

B. PROGRAM OBJECTIVES

The Lincoln Laboratory program in Restructurable VLSI is based on a concept of programmable conducting paths that can interconnect individual circuit cells both for initial circuit testing and defect avoidance, and for subsequent modification of the basic system architecture to accommodate different applications or changing user needs.[1] Although the immediate benefits of our restructuring technology will be in the areas of VLSI testing and defect avoidance, we believe that the ultimate value of the concept is that it permits physical configuration to be a programmable attribute of integrated circuitry. When combined with conventional software programmability at the cell level, restructurability provides an additional layer in a programming hierarchy that is ideally matched to a modular VLSI environment. Figure II-1 illustrates the concept in the context of a signal-processing application. Basic cells are of MSI to LSI complexity and are treated as elemental units for purposes of testing and defect avoidance. Two generic cell types are shown, i.e., memory (M) and computation (C), which in turn may be specialized or programmed for specific tasks via conventional software methods or programming of connections on control pins. After testing, good cells are interconnected through programmable conducting paths to produce a final system – in this case a 64-point pipeline FFT. The same programmable paths also provide access to individual cells for testing purposes. A chip (or wafer) will include enough cells of each generic type so that for the expected cell yield some reasonable percentage of chips can be wired up from the good cells.

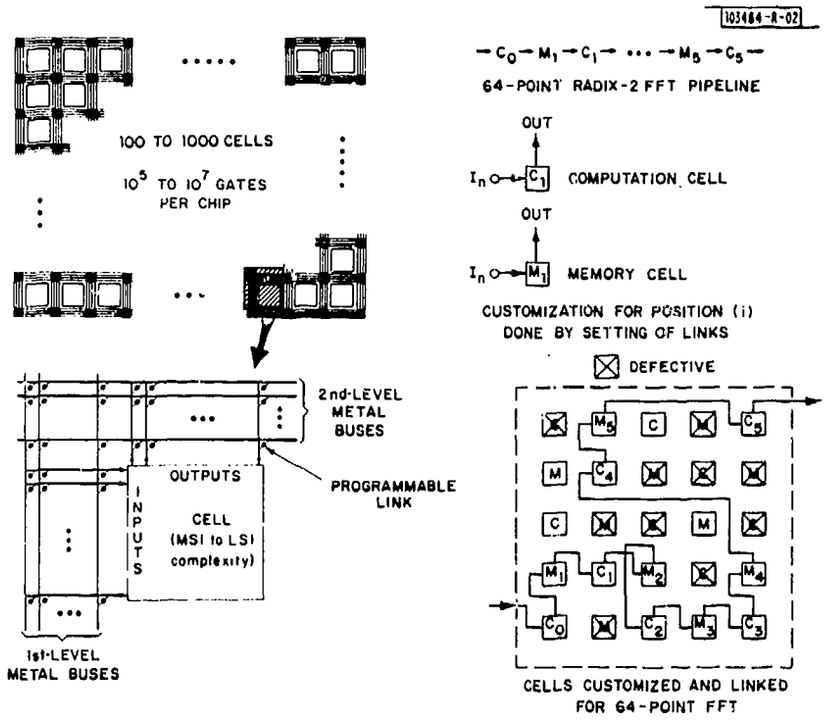


Fig. II-1. Restructurable VLSI using programmable links.

The use of restructurable logic for customizing large-scale chips or wafers for specific applications is of special interest in military contexts for which production volumes may be insufficient to amortize the initial design costs of complex VLSI chips, but for which wafer-scale integration is critical for reasons of size, weight, and power consumption. Restructurability offers the potential to tailor a single design to a variety of different system applications with resulting reductions in design time and cost for custom integrated systems. Restructurability also offers the potential for logic reconfiguration after deployment in the field. This capability could allow machine architectures to be modified dynamically to match individual processing tasks or to be upgraded in response to new system requirements.

A major long-term program objective is to develop a powerful new class of flexible/programmable signal-processing architectures matched to the modular nature of high-density large-area VLSI circuitry. Our basic concept involves the use of restructurable conducting paths for interconnecting individually programmable modular elements. Although such structures may not always be suited to general computational problems, complex signal-processing tasks can usually be decomposed into sets of relatively simple interconnected modular functions. For example, one can envision a very direct relationship between a signal-processing block diagram and the actual modular decomposition of the process in a Restructurable VLSI machine. In order to achieve this long-term objective, we require a better understanding of the functional requirements and internal architecture of the basic system modules, and we need to understand the limitations and trade-offs between various reconfigurable interconnection topologies. We also will need to develop data-processing architectures matched to RVLSI since most potential whole wafer systems, such as vocoders, will require both signal- and data-processing capability.

An understanding of the relationships between cell complexity and amount of restructurable interconnect needed for testing and defect avoidance is required. Cells must be large enough so that the ratio of logic to cell interconnect areas is reasonable, but small enough to result in satisfactory yield. Cell size also will be determined on the basis of efficient partitioning for testing. A very appealing concept is that of integrating appropriate test and reconfiguration control circuitry directly onto a wafer, and providing a self-testing and restructuring capability that allows the system to achieve operational status without external manipulation. The design of a methodology for automatic on-wafer testing is a major long-term goal of our Restructurable VLSI program.

C. TECHNICAL BACKGROUND

We have partitioned the process of Restructurable VLSI design, testing, defect avoidance, and reconfiguration into four major steps called placement, routing, assignment, and linking. Placement refers to the physical location of basic cell types on a chip or wafer. Routing is the process of designing metal paths of various lengths and orientation such that they can be used in a flexible way for custom interconnection of the various cells. After a chip or wafer is fabricated the testing process identifies usable cells and these are then associated with individual logic modules. We call this process assignment. Cells themselves may be customized after the assignment procedure; e.g., if they are programmable logic arrays, read-only memories, or nested clusters of Restructurable VLSI circuits. Finally, linking is the process of interconnecting the various metallization routes to form appropriate conducting paths between cells. The above discussion assumes that interconnect buses are partitioned into fixed lengths during fabrication. An additional degree of restructuring freedom is provided if arbitrary metal segmentation is possible after wafer fabrication (e.g., by using a laser zapper).

Since the locations of defective cells will be unique to each chip, the placement strategy must be designed such that there are appropriate numbers of each basic cell type conveniently located on the chip. The routing must provide the greatest degree of interconnection flexibility given the expected cell yield and the functional requirements of the subject design. Because of the limited ability of a given placement and routing to cope with all possible defect patterns or machine topologies, occasions will arise in which there may be sufficient numbers of functioning cells to complete the design, but a linking cannot be found to interconnect them. If the chip contains relatively large quantities of identical cells an alternative assignment scheme might be found for which a valid linking exists. The processes of assignment and linking are thus seen to be mutually interactive. A similar conclusion can be drawn for the placement and routing functions, but these are coupled through the cell yield statistics rather than by specific defect patterns.

A fundamental difference between conventional logic-driven path switching and restructurable linking is that the latter can be expected to be exercised relatively infrequently compared to the rates at which data may be switched in a given machine. This affords a wide degree of flexibility in the choice of programmable link technology, since the links need not be restricted to the same family of switching elements that constitute the bulk of the system. Candidate approaches include the use of once-programmable fusible links similar to those used in read-only memories, laser-based methods in which connections can be permanently broken or created, and a variety of transistor switching designs that require differing amounts of additional circuitry for programming purposes and/or for nonvolatile retention of state information.

An important goal in our research is to assess the utility of the various programmable linking approaches for testing, defect avoidance, and system reconfiguration. Each of these imposes a different set of requirements and constraints on the types of links that can be used. We expect that the best compromise between system flexibility and complexity of the restructuring circuitry will be achieved through the use of more than one linking technique in a given design. For example, if the objective were simply defect avoidance or initial customization of the machine architecture, then restructuring would only occur in the last stages of the manufacturing process and links would need to be programmed only once. The simplicity, electrical efficiency, and nonvolatility of fusible or laser-programmed links are well matched to the customization and defect avoidance requirements. On the other hand, testing might require that conducting paths be reprogrammed several times in order to gain connectivity to various cells in time succession, but link nonvolatility would not be required. Flip-flop controlled AND gates using the same circuit technology as in implementing cell logic can provide such links to test nodes. These links require additional chip area for control paths and storage of state information and would therefore be used sparingly.

The concept of architectural reconfiguration is best realized with links which are both nonvolatile and reprogrammable throughout the lifetime of the chip. This application has motivated us to explore new linking methods in which the required features can be achieved using small geometry, easily programmed devices. Our work in nonvolatile MNOS links is directed at this problem area.

III. RESTRUCTURABLE INTERCONNECT

A. INTRODUCTION

Restructurable interconnect is a new technology which is crucial to the viability of RVLSI. Restructuring of cell interconnect is done through an array of metal buses and programmable links. The buses and links must be usable for test access, defect avoidance, reconfiguration, and customization with acceptable electrical performance and minimum consumption of area and power. Good yield is essential since it may be difficult to make substitution for defective links and buses. The important functional characteristics of a link are whether it can be programmed more than once; and if it is reprogrammable its speed of response to programming signals and whether it holds its programmed state without power.

The functions for which links are used impose differing requirements on them as shown in Table III-1. Two forms of reconfiguration are listed: "dynamic" must be done in a time comparable (within several orders of magnitude) to the system clock period while "occasional" describes reconfiguration which can be done more slowly. Some measure of reprogrammability

TABLE III-1
FUNCTIONAL REQUIREMENTS ON LINKS

	Reprogrammability	Speed of Programming	Nonvolatility
Defect Avoidance	No	–	Yes
Customization	No	–	Yes
Test Access	Yes	Slow	No
Dynamic Reconfiguration	Yes	Fast	No
Occasional Reconfiguration	Yes	Slow	?

may be required for defect avoidance where the final testing can be done only with programmed interconnect. A combination of once-programmable normally open and normally closed links may suffice. It is assumed that the connectivity information for dynamic reconfiguration is stored separately (on or off the wafer) so that link nonvolatility is not required. The requirement for nonvolatility for occasional reconfiguration is application dependent.

The three types of links which are being developed are described in the following sections.

B. ELECTRICALLY REPROGRAMMABLE NONVOLATILE LINK

The ideal link is nonvolatile and electrically reprogrammable an unlimited number of times. An electrically alterable resistor such as can be built with a chalcogenide material is one possibility, but such a two-terminal device presents problems in isolation between the control and signal lines. An MOS pass transistor with charge storage in the gate dielectric does not have this problem. Either MNOS or FAMOS devices can be used in this way. Both the resistor and pass transistor are bidirectional devices but their series resistance may seriously limit speed.

Figure III-1 shows a link comprising an MNOS transistor and a bipolar transistor. The MNOS transistor is used as a pass transistor and the bipolar transistor provides power gain. The bipolar transistor introduces voltage shift on the signal line which limits the number of links which can be connected in series and the unidirectional nature of this link complicates the design of the restructurable interconnect. These devices have the disadvantage of requiring special fabrication steps and control voltages (30 V) larger than normally used with logic circuits. Figure III-2 shows how the MNOS and bipolar transistors can be merged to minimize link area. This link is being fabricated along with a test section of a programmable logic array where these links are used to make an electrically alterable PLA. In the PLA unidirectionality is not a disadvantage and the bipolar device allows fast switching on the heavily loaded lines in the AND and OR arrays. The link is about $(10\lambda)^2$ in size with $\lambda = 3$ μm in this implementation. A circuit variation described in the literature[2] which uses low voltages for selection and an unswitched high voltage has been examined as a possible alternative to the current approach.

C. ONCE-PROGRAMMABLE LINKS

The simplest link is a once-programmable connection such as the fusible link used in PROMs. We call this link, where programming removes a connection, deletive. In an additive link programming makes a connection. We propose to program both types with a laser beam. Removal of metal with a laser is done routinely to correct defects on IC chrome masks. Addition of a connection could be done by melting through the insulation at the crossing of two metal lines, annealing of amorphous silicon to change it to a low-resistance connection, or formation of a connection in a chalcogenide. Experiments are in progress on forming connections in two-layer metal.

The laser scheme reduces chip complexity since no on-chip access circuitry is required but can be used only prior to packaging. Once-programmable links have excellent ON/OFF resistance ratios and small size and therefore could be used to program the segmentation of interconnect metal which would greatly simplify the routing of signals on a chip. Where a limited amount of reprogrammability is required, additive and deletive links can be combined. Ideally, this link does not add any area to the interconnect structure because the metal lines themselves are being connected or opened but second-order effects will undoubtedly dictate that some additional area be used.

D. ELECTRICALLY REPROGRAMMABLE VOLATILE LINKS

A logic gate controlled by one bit of storage is an electrically alterable volatile link. It can be switched at high speed and provides signal gain. Fabrication of this link is identical with fabrication of standard logic devices. Of the links considered this one consumes most chip area and power; however, for cases where a number of signal lines can be controlled as a group one FF can control multiple gates, thereby saving space and reducing access circuitry for link control. A circuit comprising a CMOS FF and tri-state gate circuit was designed and laid out in a form suitable for implementing such a volatile link. The area of the link was about $(55\lambda)^2$ with $\lambda = 3$ μm.

E. COMPARISON OF LINKS

Table III-2 presents a comparison of the three types of links where the reprogrammable nonvolatile column characteristics are representative of the merged MNOS-bipolar device. For

TABLE III-2
COMPARISON OF THREE LINK TYPES

	Reprogrammable, Nonvolatile	Once-Programmable	Reprogrammable, Volatile
Area	$(10\lambda)^2$	Small	$(55\lambda)^2$
Power Consumption	Smaller	None	Larger
Access and Control Circuitry	High voltage	None	Standard logic
Programming Speed	Slow	–	Fast
Bidirectional	No	Yes	No
Fabrication	Special	Standard (?)	Standard logic
Signal Transmission	Unidirectional, gain	Bidirectional, low impedance	Unidirectional, gain

defect avoidance and customization before packaging, the once-programmable link is superior to the others. Results of ongoing device development will determine if a reprogrammable nonvolatile technique is viable.

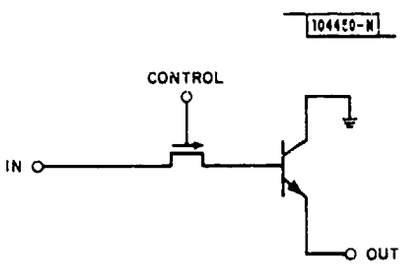


Fig. III-1. Nonvolatile programmable link – circuit.

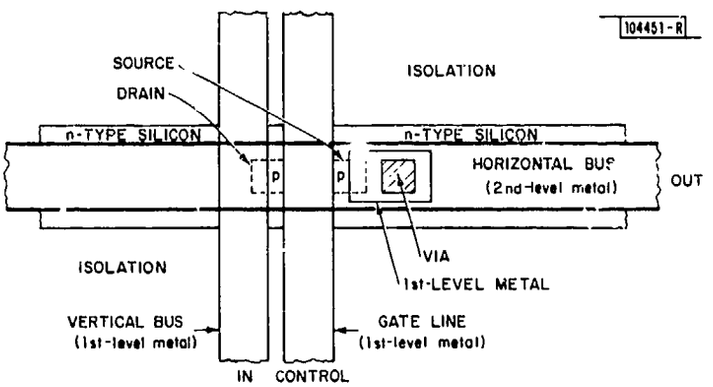


Fig. III-2. Nonvolatile programmable link – plan view.

IV. DESIGN AIDS FOR RESTRUCTURABLE VLSI

A. INTRODUCTION

Our approach to the design of a Restructurable VLSI system has three phases: (1) system and logic design with emphasis on testability and partitioning into cells, (2) physical design of the chip, and (3) final restructuring of the fabricated chip based on test results. In each phase there are aspects of the design process unique to Restructurable VLSI. During this period we have developed a hardware description language especially well suited to design with collections of cells which should assist and encourage the designer to think in these terms. This language and development of a translator are described in the next section. An important problem in the restructuring process is that of finding signal paths between good cells on the interconnect provided. This is called the linking process because the solution specifies which links to turn on or off. Development of a research linker is described in Sec. C. Work on the assignment problem, that is, binding of logical cells to good physical cells, is described in Sec. V in the context of specific RVLSI applications.

B. HIERARCHICAL AND ITERATIVE STRUCTURE DESCRIPTION LANGUAGE (HISDL)

1. Introduction

The specification of interconnections of components in a system of VLSI complexity and the subsequent transformation of those specifications to connection or net lists are tedious and error prone when done manually. When a language that is sufficiently rich in constructs for the formal description of such specifications is used, then the description is easily written by the designer and the generation of connection lists can be done automatically. For Restructurable VLSI we need a language for the description of hierarchical and iterative designs. At present, description of behavior is of secondary importance.

Many computer hardware description languages have been developed.[3,4] Until recently the emphasis in these languages has been the description of the behavior and register level structure of a computer. Some languages that facilitate the specification of interconnections of components (i.e., networks) at a high level have been proposed and are being or will be implemented. For example, AHPL III (Ref. 5) is an extension of AHPL (Ref. 6) to handle the specifications of structures for describing a network of MSI parts. However, it is still a behavior-oriented language not well suited to explicit interconnect description. Another language that has been studied as a candidate for the structure description language is CASL (Ref. 7). However, CASL is currently being implemented and the present emphasis is not the generation of connection lists. A third language, SDL (Ref. 8), supports hierarchical descriptions of structures. However, SDL does not have constructs for specifying replication of interconnections of substructures. SCALD (Refs. 9-11) is a graphics-based hierarchical digital logic design system which does generate wire lists. The drawback with this system is that it requires graphics terminals as input devices and it is unclear whether arrays of structures of more than one dimension can be conveniently specified as a structure.

No structure description language with textual input that meets our needs is available. Therefore a language has been defined and we are writing a translator which generates wire lists. The language facilitates the hierarchical description of interconnection of structures and also allows the user to specify replication of structure interconnections. Due to these two main

features of the language, it is given the name HISDL (Hierarchical and Iterative Structure Description Language). The language is designed strictly for the description of interconnections of structures and is oriented to the design of a network of components which may be of LSI complexity. Structures are modular, in the sense that components are viewed as copies of some user-specified structure types. HISDL does not describe the behavioral properties of structures although it could be used to describe interconnection of cells with known behavior and thereby serve as input to a simulator.

Section 2 describes the features and constructs of HISDL together with an illustrative example. The implementation details of version 1 of the HISDL translator and the current state of implementation of the translator are given in Sec. 3. From the experience gained thus far, some plans for modifications, extensions, and enhancements of the translator have been developed and are described in Sec. 4. The syntax of HISDL is given in an appendix.

2. Language Description

The system to be described in HISDL is viewed as a hierarchy of interconnections of components. Each component is connected to other components at the same level by connecting the appropriate ports of the structures together. The details inside each component at a given level in the hierarchy need not be known when the connection is made. It is the user's responsibility to ensure that the connections are compatible with the behavior of the structure.

A component is a copy of a structure type which serves as a template for making the copy. There is always a name and a structure type associated with a component. Since a component can be part of an array of components, its name can have array subscripts. Each structure type is defined by a structure type definition. Figure IV-1 shows the block diagram representation of a structure (a 16-bit adder) and its HISDL description is given in Fig. IV-2. A structure type definition has a header (line 1 of Fig. IV-2) which contains the keyword STRUCTURE followed by the structure type name and a list of parameters. These parameters are the names of the I/O ports of the structure type. Following the header are the I/O declaration lists (lines 2 and 3). Line 2 declares that the structure type has two input ports called AUGEND and ADDEND, both with path widths of 16 and the individual lines of the ports are numbered 0 to 15. Line 3 declares that the port, SUM, of path width 16 is an output port. The list of components used in this structure type is given in line 4 which declares that the components are ADDHIGH and ADDLOW of structure type ADDER8. Lines 5 to 15 define the interconnections of the components. Each line is a connection list representing a set of ports to be connected together. Each port has a component name (optional) followed by a port name. The component and port names of a port are separated by a period. If the component name is absent, then the port belongs to the structure type being defined. Line 16 terminates the structure type definition. The 8-bit adder, ADDER8, is defined to be a cell type. Logically, a cell type is the lowest-level structure type in the structural hierarchy and the special keyword CELL is used. There need not be any distinction between cells and structures as far as the logical connections are concerned. However, it is prudent to provide flexibility for specifying structures that are predefined and their descriptions exist in a library. Such structures may represent the lowest level in the hierarchy that the user is interested in.

One of the important features of the language is the FOR construct. With this construct the user can easily specify the interconnections of an array of components. The maximum number of dimensions of an array in HISDL is four allowing one to specify the structure of a four-dimensional array of components.

Connections of structures are specified as lists of ports. Each list can optionally have a user specified name. A connection list is a list of ports delimited by special characters like "/" and braces. Besides using the connection list, connection of components can be specified as component invocations as illustrated in Fig. IV-3 which describes the same structure given in Fig. IV-1. A component invocation is very similar to a procedure call in a general programming language. By specifying the name of a component followed by a list of ports which has a one-to-one mapping to the parameter list of the component's structure type, the user can connect structures together. The translator will generate the connection lists from the component invocations provided.

A typical HISDL program is a list of structure or cell type definitions. One of the structure type definitions is the highest level or root structure representing the starting structure type. The translator will use the root structure to obtain the net lists for the whole structure defined. The net list for the whole program is generated after all structure and cell type definitions are expanded. In the first version of the translator, the first structure or cell type definition encountered in the HISDL source program is the root structure.

3. Implementation of the HISDL Translator (Version 1.0)

The HISDL translator currently being implemented can be broadly divided into two parts – the parser and the data base. A language design tool, LANG-PAK (Ref. 12), is used to implement the translator. The syntax of HISDL is first defined in the LANG-PAK meta-language which is similar to the Backus-Naur Form (BNF) of language specification. The meta-language allows one to specify codes for semantic actions to be taken as the various constructs are parsed and processed. The semantic codes are "compiled" by a subroutine called the semantic compiler into operation codes to drive the semantic machine, another subroutine. The interface between the parser and the data base is the semantic machine which is coded to call the appropriate data-base subroutines as the HISDL statements are processed. All the LANG-PAK related subroutines have been implemented and a test version of the translator for syntax checking purposes is currently running under VM/370 CMS. The version of LANG-PAK that is installed is a FORTRAN version.

The translator creates a data base as it parses the input statements. The data base is an internal representation of the structure described by the HISDL program. It is used to generate the connection lists. About ten FORTRAN arrays are used to implement the data base at the current stage. A master directory is used for accessing the data base using names, i.e., structure/cell type names, component names, port names, names of connection lists as well as names of FOR variables. The subroutines for storing the I/O and component declaration lists information have been implemented. There is a table for each structure/cell type defined. Another table contains information for all components declared. The entries in the tables are pointers to other tables (including the tables themselves). All I/O port names and path widths are stored in common arrays and pointers are used to access the right list (or lists) of I/O port names for the given structure/cell type or component. It is common for the ports in a structure definition to have the same path width specifications. Thus to save space, all path width specifications (as well as array dimension specifications) are stored in a common storage area and each entry in the area is a unique specification. Thus it is possible to share these specifications among many ports as well as array names. This is true, also, for names; that is, all names are stored in a common area and the entries are all unique. Thus if there are two instances

of the same name being used, the appropriate entries for the name contain pointers to the same location in the same common area.

Work is in progress in implementing the data-base subroutines for storing information on connection lists and component invocations and in the development of an algorithm for generating the net lists using the information in the data base. The organization of the connection list and component invocation information in the data base is dependent on the net list generation algorithm. The output of the translator will initially be the net lists and tables of component and port names. The tables are used in conjunction with the net lists to aid the user in associating the net lists with logical component and port names of the program.

A number of HISDL programs were syntactically checked using the test version of the translator. For a program of 135 lines, the CPU time was 2.66 s for the translator running under CMS on an Amdahl 470/V7. Besides syntax checking, a listing of the program was generated together with a special file. The file was used for debugging purposes and contained the contents of the data-base arrays and the actual parameters of the data-base subroutines called.

4. Modifications, Extensions, and Enhancements of HISDL

The experience gained in the use of the test version of the translator has provided some insights into extensions and enhancements of HISDL. One proposed modification to the language is the replacement of the list of port names in the structure type definition header with the I/O declarations. This removes the redundancy that now exists between the port names in the I/O declarations and the list of port names in the structure type definition header and is consistent with CONLAN (Refs. 13-15).

For more efficient specification of arrays of interconnected structures, a conditional construct like the IF...THEN... ELSE should be provided. This addition to HISDL will make it easier for the user to specify special connection conditions, for example at the edges of an array of interconnected structures. The conditional construct when used in conjunction with connection list specifications will allow the inclusion or exclusion of certain connection lists when the specified condition holds (or does not hold). In the first version of HISDL, the special connection at the edges of an array of interconnected structures can be specified using multiple FOR constructs.

Other language extensions may be desirable. For example, the translator can be extended to print the hierarchy of the structure described or to generate its graphical equivalent. Furthermore, the translator can be modified to create a more permanent data base which can be used over and over again. This will allow the user to "build" up a system description from many different HISDL program runs, each run describing a part of the system. With this capability, users can share the common data base.

C. RESEARCH LINKERS

1. Introduction

After a RVLSI wafer is tested and an acceptable assignment is made, the desired cell-to-cell connections are made by setting links. The process that finds the paths for all the specified connections on a previously defined pattern of interconnect and links is called linking. It is analogous to the process of routing in the design of ICs and PCBs.

A linker operates in a world of pins and segments. Pins are the I/O ports of a cell. A segment is a physical conducting line that is laid out on a channel, which is the space between

rows of cells. A segment does not need to run the full length of the channel; several short segments may be collinear. The overall connectivity possibilities of the interconnect pattern depend on the density and length of the segments, as well as where links are located. A linker operates on two data bases. The first is the pin-to-segment and segment-to-segment connectivity information. The second is the list of pins to be connected.

Our current linkers are experimental or research linkers. They are being used to investigate both linking heuristics and to study the effectiveness of different interconnect patterns for certain applications. This use as a research tool demands the ability to work with a variety of link and segment characteristics, which in turn requires a variety of cost functions. In order for the cost function to be easily modified it is separated from the search algorithm. This allows all paths to be found followed by application of the cost function to find the least cost path. Typical items that would be included in the cost of any path would be the total number of active links on the path, the maximum number of links in series in a path, the physical length of the path, and the number of links made unusable for future paths. The cost of all these items would be technology-dependent.

We have programmed and are operating three research linkers. They are an exhaustive linker, a cluster filter followed by the exhaustive linker, and a graph reduction linker. All are written in PL/1 and are running on the Amdahl 470/V7. Connectivity information can be extracted from either a Calma layout or a textual description of the positioning of cells, segments, and links. The linkers, as written, find solutions for multipin nets through bidirectional links. If unidirectional links were used only small changes to the present data structures and search algorithms would be required.

2. Exhaustive Linker

The exhaustive linker finds all the possible solution paths of a specified connection and applies a cost function to select the best path. In this method there is a clear separation of the path search and the application of the cost function in agreement with the goals of a research linker.

For each net, the exhaustive linker does a depth first search through the various path possibilities by expanding from an arbitrarily chosen pin through any valid combination of segments until the path contains all the pins in the net. The path's cost is measured at this point and is kept if it is smaller than the current minimum cost path. The search continues through different path permutations until there is no possibility of finding a cheaper cost path or until an arbitrary cutoff point is reached.

Although the exhaustive linker will find all the possible paths, it does so at high cost. We experienced a growth in time required to find an optimal solution that was exponential with the number of pins in a net and the maximum number of links in series from one pin of the net to another pin. The following tabulation shows this growth:

Pins	Links (series)	Time (s)
2	2	4.0
3	2	4.9
2	4	10.0
3	4	21.4
:	4	282.0

3. Cluster Filter

The cluster filter was developed to reduce the number of segments and thus the search space of the exhaustive linker. Those segments that have no chance of being in a minimum link solution path are eliminated from further consideration and the resulting group of valid segments is then passed to the exhaustive linker.

The cluster filter begins with all the pins of the current net considered as "roots" of separate equivalence classes. For each pin, a one-link expansion finds all the available segments and pins that can be reached from that pin by going through one link and these segments and pins are placed into that pin's equivalence class. This one-link expansion proceeds for each equivalence class in turn until two or more classes are found to have some segment in common. The Venn-like diagram of Fig. IV-4 helps to illustrate the situation. The two large circles represent two equivalence classes and contain many segments and pins. The smaller "root" circles are subsets of the equivalence classes. The paths are a group of segments that connect the "root" to the common segments. These paths usually contain only a small portion of the set of segments and pins that make up the equivalence class. At this point the segments in common, the "root" of each intersected equivalence class, and the paths are merged or clustered to form a "root" of a new equivalence class. Those equivalence classes that did not intersect on the last iteration continue to expand. The process of expansion continues until one equivalence class is formed that contains all the pins. That set is then passed to the exhaustive linker to find the minimum cost path.

Figure IV-5 shows a simple layout on which pin 1 of cell 00 (00.1), pin 2 of cell 01 (01.2) and pin 3 of cell 13 (13.3) are the pins of a net. Following the procedure outlined above, pins 00.1, 01.2, and 13.3 are made the "roots" of their own equivalence classes A, B, and C, respectively. After a one-link expansion A contains pin 00.1 and segments 1, 3, and 5; B holds pin 01.2 and segments 1, 2, and 6; and C holds pin 13.3 and segments 8, 10, and 12. Since A and B have segments 1 and 3 in common, they are merged. Segments 1 and 3 with pins 00.1 and 01.2 form the "root" of the new equivalence class AB. There are no segments in common with C at this point. Another one-link expansion adds segments 5, 6, 13, 14, and 15 to AB and segments 15 through 18 to C. Segment 15 is common to AB and C. The path from segment 15 to the "root" of C is through segment 12 and there is no path to the "root" of AB, since it already touches the "root." The new equivalence class ABC is formed containing all three pins and segments 1, 3, 12, and 15. The exhaustive linker need only consider four segments when used with the filter, vs eighteen segments without the filter. In larger layouts the savings are even greater.

An implied cost criterion was introduced into the search process by this heuristic filtering algorithm, namely that only paths with the minimum number of links would be minimal cost. This reduction in flexibility bought a large gain in speed. Use of the cluster filter followed by the exhaustive linker made an order-of-magnitude improvement in speed in most cases, but the time needed still grew exponentially as shown in the tabulation below.

Pins	Links (series)	Time (s)
2	2	2.9
3	2	3.0
2	4	3.8
3	4	5.4
4	4	5.1
4	6	22.6
5	6	89.5

4. Graph Reduction Linker

The graph reduction linker reduces the problem complexity even further and therefore achieves impressive time reductions over the other linkers. It takes a group of segments and pins, presently the output from the cluster filter, and forms a directed graph. The pins and segments are the nodes in the graph and the links connecting them are the arcs of the graph. The graph is built by starting at an arbitrary root pin, and traversing through links and segments away from the root pin until all the segments and pins are in the graph. A graph built this way could conceivably be quite complicated with many possible paths that connect the pins. For example, Fig. IV-6 shows a partial layout in which the segments that have survived the cluster filter have been darkened. A graph built from these segments starting from pin 01.1 looks like Fig. IV-7. Those segments or pins with unique connectivity possibilities are represented by circular nodes, while those segments with identical connectivity are grouped together in rectangular nodes.

This graph can be reduced and a solution found by recognizing three things. First, pins are always needed in any solution path and therefore are exempted from the search process. Second, at a specific level in the graph some segments may have identical connectivity possibilities; i.e., touch the same segments, and/or pins, and can be reduced to one minimum cost segment. For example, in the graph of Fig. IV-7, segments 37, 38, 41, 44, and 45 have identical connectivity. However, in the full layout it is found that segments 37 and 38 cost less by any cost criterion than any of the others. One of the two is chosen, since they are equivalent. By applying the same procedure to every rectangular node in the graph, the example graph is reduced from 35 possible nodes to 15 nodes. The third thing to notice is that certain segments are always needed in any solution path; these can be seen as "constrictions" in a graph. In the example this would happen if all the segments in the two rectangular nodes with segments 70 and 37 had been previously allocated by the linker. Then the minimum cost segment in the node with segment 48 would be a constriction. The graph reduction linker takes this into account by not using this segment in the enumeration process. This divides the graph in two, which allows the linker to consider the smaller problem of finding optimum paths through the subgraphs.

By eliminating some segments from consideration and requiring others to be in the solution, the graph reduction linker can often substantially reduce the graph. Then simple enumeration of all possible paths can be done in a reasonable amount of time. This allows most nets to be linked in a fraction of a second, even if the routes taken are tortuous or the net is multipinned. Some nets, however, cannot be reduced very much and then one sees the customary exponential time growth. However, the constant is not as big as in the other linkers, because the enumeration or search process is more efficient. This linker has no implied cost assumptions.

The graph reduction linker is currently being used to test the interconnect pattern of several layouts. In one experiment a 128-cell layout used for the integrator was tested. Fifty-six nets were linked in 10 s. This experiment proved the worth of both the linker and the integrator's interconnect pattern.

Successful linking may be dependent on the ordering of nets and/or the assignment of cells on the chip. Unless the interconnect pattern is designed with linkability in mind, such that one net's linkage does not adversely affect other nets' linkages, the linker may have to be modified in a couple of ways. Besides presorting the net list, the linker might need the ability to unlink nets that prevent other nets from being linked. Another modification would reflect the fact that the assignment and linking processes are interdependent. In the case that no complete linking could be done for the current assignment, the assignment program would have to do another assignment using information derived from the linking failure.

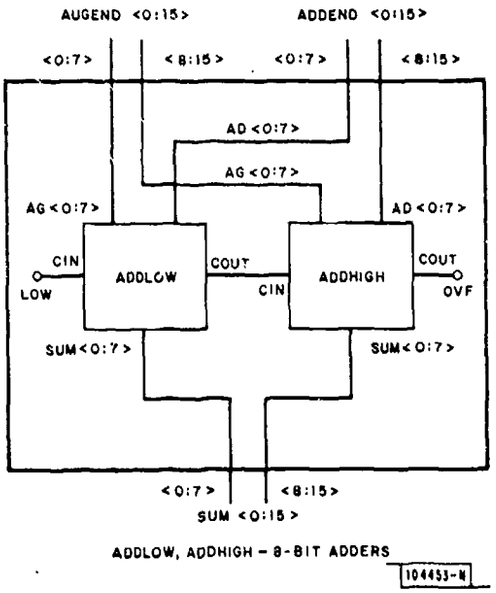


Fig. IV-1. Structure of a 16-bit adder.

```
                %
                %  16-BIT ADDER DEFINED IN TERMS OF
                %  CONNECTION LISTS
                %
1               STRUCTURE ADDER16 (AUGEND,ADDEND,SUM)
                   % DEFINITION OF A 16-BIT ADDER STRUCTURE
                   % THAT USES TWO 8-BIT ADDERS.
2                  IN AUGEND<0:15>,ADDEND<0:15>
3                  OUT SUM<0:15>
                   % STRUCTURE USES TWO 8-BIT ADDERS
4                  COMPONENTS ADDHIGH,ADDLOW : ADDER8
                   %
5                  BEGIN
                      % ADDLOW INPUTS
6                     /ADDLOW.AG,AUGEND<0:7>/
7                     /ADDLOW.AD,ADDEND<0:7>/
                      % LOW 8 BITS OF SUM
8                     /ADDLOW.SUM,SUM<0:7>/
                      % ADDHIGH INPUTS
9                     /ADDHIGH.AG,AUGEND<8:15>/
10                    /ADDHIGH.AD,ADDEND<8:15>/
                      % HIGH 8 BITS OF SUM
11                    /ADDHIGH.SUM,SUM<8:15>
                      % CONNECT CARRY-IN OF ADDLOW
                      % TO INTERNAL PORT LOW.
12                    /ADDLOW.CIN,LOW/
                      % CONNECT CARRY-OUT OF ADDHIGH TO
                      % INTERNAL PORT OVF.
13                    /ADDHIGH.COUT,OVF/
                      % CONNECT CARRY-OUT OF ADDLOW TO
                      % CARRY-IN OF ADDHIGH
14                    /ADDLOW.COUT,ADDHIGH.CIN/
15                 END
                   %
16              ENDSTRUCT
                %
                % DEFINE 8-BIT ADDER
                % AG, AD, SUM ARE THE 8-BIT AUGEND, ADDEND
                % AND SUM RESPECTIVELY.
                % CIN AND COUT ARE THE 1-BIT CARRY-IN AND
                % CARRY-OUT RESPECTIVELY
17              CELL ADDER8 (AG,AD,SUM,CIN,COUT)
18                 IN AG<0:7>, AD<0:7>,CIN
19                 OUT SUM<0:7>,COUT
20              ENDCELL
```

R; T=0.01/0.07 13:42:16

104454-S

Fig. IV-2. HISDL description of the 16-bit adder.

```
        %
        %   A 16-BIT ADDER USING COMPONENT INVOCATION
        %
  1     STRUCTURE ADDER16 (AUGEND,ADDEND,SUM)
              % DEFINITION OF A 16-BIT ADDER STUCTURE
              % THAT USES TWO 8-BIT ADDERS.
              IN AUGEND<0:15>, ADDEND<0:15>
  3           OUT SUM<0:15>
  4           COMPONENTS ADDHIGH,ADDLOW :ADDER8
              %
  5           BEGIN
                 % INVOCATION WITH CONNECTIONS OF ADDHIGH
  6              ADDHIGH(AUGEND<8:15>,ADDEND<8:15>,SUM<8:15>,
  7              ADDLOW.COUT,OVF)
                 % INVOCATION WITH CONNECTIONS OF ADDLOW
  8              ADDLOW(AUGEND<0:7>,ADDEND<0:7>,SUM<0:7>,LOW,INT)
  9           END
 10     ENDSTRUCT
          %
          % DEFINE 8-BIT ADDER
          % AG, AD, SUM ARE THE 8-BIT AUGEND, ADDEND
          % AND SUM RESPECTIVELY.
          % CIN AND COUT ARE THE 1-BIT CARRYIN AND
          % CARRYOUT RESPECTIVELY
 11       CELL ADDER8 (AG,AD,SUM,CIN,COUT)
 12          IN AG<0:7>, AD<0:7>,CIN
 13          OUT SUM<0:7>,COUT
 14       ENDCELL

R; T=0.01/0.05 13:43:30
```

104455-S

Fig. IV-3. HISDL description of the 16-bit adder with component invocation.

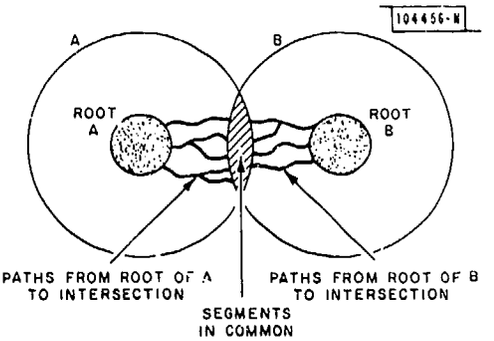


Fig. IV-4. The intersection of two equivalence classes.

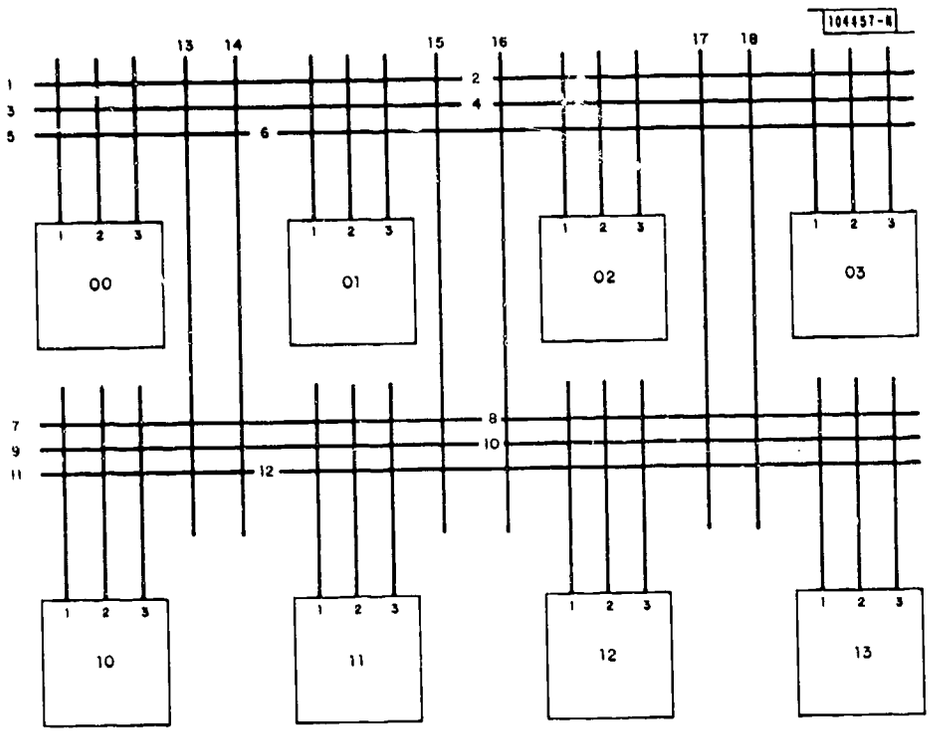


Fig. IV-5. Eight 3-pin cells with interconnect.

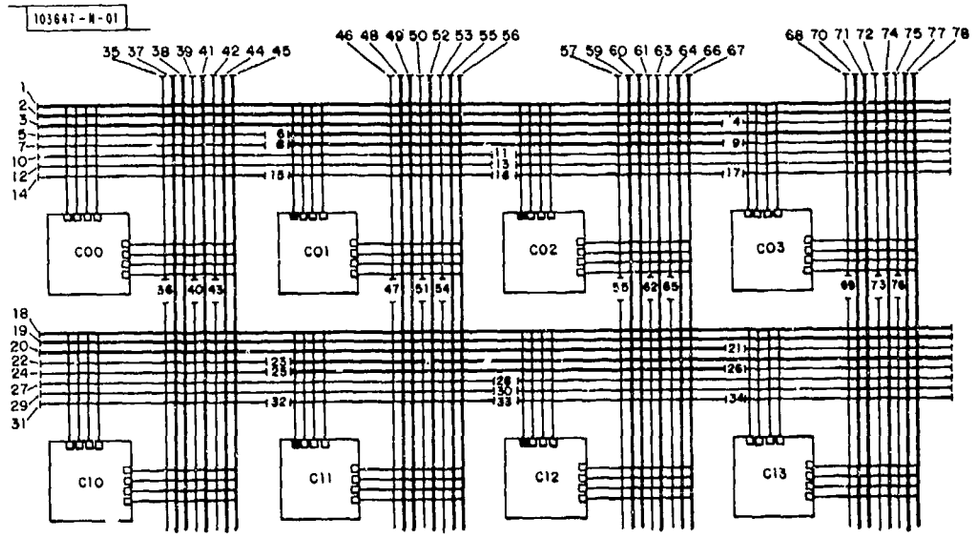


Fig. IV-6. Eight 8-pin cells with interconnect.

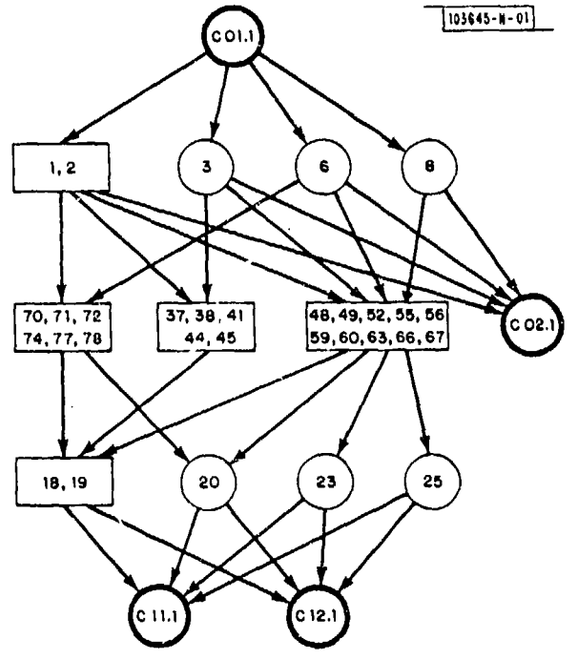


Fig. IV-7. Graph for a net on layout of Fig. IV-6.

V. APPLICATION OF RESTRUCTURABLE VLSI

A. INTRODUCTION

Advances in implementation of digital systems have always influenced system architecture, the outstanding example being the effect of decreasing cost of memory. We have begun to investigate ways in which the dramatically higher levels of integration achievable with RVLSI will influence the way we think of systems. The physical partitioning of RVLSI into cells suggests a good match with cellular architectures and the uniformity of a systolic array is attractive for RVLSI implementation. Section B describes how a regular array might be restructured for defect avoidance. We plan to test our concepts and design tools on a series of increasingly challenging applications. A digital integrator for a spread-spectrum packet radio receiver has been selected as the first application. The system and early design results are described in Sec. C.

B. SYSTOLIC ARRAYS

Architectures with a small number of cell types and mostly local interconnect seem advantageous for implementation in Restructurable VLSI. One such architecture is the systolic array which is a cross between a pipeline and a single-instruction, multiple-data (SIMD) stream machine. Like a pipeline, each entity in the array repetitively receives data from its neighbors, operates in a fixed manner on the data, and transmits the transformed data to other neighbors. Like a SIMD, each identical entity executes the same instruction.

Systolic arrays of cells with 3, 4, and 6 faces are described in the literature.[16] All of these layouts can be mapped onto an 8-faced cell layout by some logical to physical face assignment. The layout of octagonal cells with nearest neighbor connectivity shown in Fig. V-1. By restricting the number of usable faces to 6 per cell and assigning them as shown in Fig. V-2, the cell array can be made isomorphic to a 6-faced (hexagonal) cell array. Triangular and rectangular arrays can likewise be formed. When a systolic array is mapped onto an imperfect physical array then the required connectivity is not strictly local. A link can be inserted into each line in Fig. V-1 so that by opening links a cell is isolated. In order to route signals around defective cells a redundant rectangular grid of lines can be added. Figure V-3 shows routing of a signal around a bad cell in the horizontal direction. From left to right the signal passes through three normally open links which have been closed.

The redundant grid and the links, however, cannot by themselves repair the logical 3-, 4-, or 6-faced layout. Cells and their logical faces must also be reassigned. In order to investigate the effectiveness of this scheme of face reassignment and reprogrammable interconnect an assignment algorithm was programmed that would work on a flawed physical array. Simulations were done to determine what $n \times 8$ logical array would fit on an 8×8 physical array for a cell yield of 0.81. With logical arrays of triangular, rectangular, and hexagonal connectivity the average value of n for 20 trials was 4.5, 4.5, and 2.3, respectively. Therefore, with an average of 51.8 available cells, the average utilization for three cases was 36.0, 36.0, and 18.4 cells. We would expect higher utilization if the array could be configured to an $n \times n$ array, but at the expense of a more complex assignment program. One general result that could be inferred from the simulation is that cell utilization in a local only (or local mostly) network is very sensitive to the pattern of defects.

C. INTEGRATOR

In a spread-spectrum digital communication system a correlation is performed between the received signal and a copy of the code used at the transmitter. The information is in the polarity of the correlation peak. In the presence of noise it may be necessary to perform integration over a number of repeated transmissions of each bit. Since, in this case, the position of the correlation peak is not known, the integration must be performed at a number of positions or bins around the expected peak time. A combined surface-acoustic-wave (SAW) correlator and integrator has been designed and demonstrated at Lincoln Laboratory. Improved performance is expected from the combination of a SAW correlator and a separate digital integrator.

Figure V-4 is a diagram of such a system. The analog correlator output is converted to a 200-Mbps bit stream. These bits are grouped into frames of 1024 bits. The integrator samples a 256-bit window of each successive frame arriving within the integration period, and sums the corresponding bits of each frame. At the end of an integration period, which may vary from 10 to 1024 samples, the 10-bit summations for each of the 256 bit-positions within the sample window are read out.

Figure V-5 is a functional block diagram of the integrator. The sequence of operations for the integrator is as follows. An initial preset value is placed on the bus and loaded simultaneously into all 256 counters by pulsing the LOAD line. For each frame within the integration period a sample window of 256 bits is shifted into the count-enable register. After each sample is shifted into place a COUNT pulse is given which simultaneously increments those counters with a one in their bit of the count-enable shift register. At the end of the integration period the contents of all 256 counters are simultaneously transferred to 256 readout buffer registers. Thus the counters are free for a new integration period while the data from the previous period are read out. Readout is performed by shifting a single bit down the read-enable shift register which will enable successive readout buffer registers onto the bus.

Figure V-6 shows the partitioning of the 256-counter array into cells, each containing four 10-bit counters. Within each 8-cell, 32-counter column both the read-enable and count-enable shifts registers are daisy chained together in a next-neighbor type of interconnection. All other bus and control signals are routed in parallel to all cells on the wafer. We desire to choose a routing and a corresponding assignment strategy to map this logical design onto a physical wafer. The broadcast and bus lines pose no particular problem. The daisy chain lines, however, limit the types of assignment strategies usable with a reasonably small interconnect pattern while preserving the capability for defect avoidance.

An assignment strategy suitable for the class of designs using one-dimensional next-neighbor interconnect, like the integrator, has been devised. It is based on the two concepts of skipping defective cells along a column and migrating extra cells from one column to an adjacent one. Suppose we wish to implement an M*N logical array of cells on a larger X*Y physical wafer. We specify two parameters of the strategy, namely SKIP(K) and MIGRATE(L). We then assign each logical column to a physical column which has both a sufficient number of working cells and that has no more than K consecutive defective cells to skip over. If either condition cannot be met, connection can be made to an extra cell in a neighboring column up to L columns away.

A tentative routing layout has been worked out for the SKIP(1) MIGRATE(1) case which is expandable for the general case. On a wafer with fixed segmentation each vertical channel is required to have space for 2K + 2 signal lines, while each horizontal channel is required to have

space for 2L + 2 signal lines in order to implement each restructurable daisy chain line. If a technique for arbitrary segmentation at wafer probe time is available, the vertical channel width can be reduced to K + 1 tracks. The SKIP(1) MIGRATE(1) routing therefore requires four vertical and horizontal tracks per channel using fixed segmentation and two vertical and four horizontal tracks per channel using arbitrary segmentation. These routings are able to handle all fourteen cases of physical next-neighbor connections that arise in SKIP(1) MIGRATE(1). A more restricted routing can eliminate the rare interconnect cases to sacrifice some yield for lower interconnect area. It should be noted, though, that for the full routing the SKIP(K) MIGRATE(L) is only a sufficient condition for assignability and not a necessary one. There are many assignments which though they are not strict SKIP(K) MIGRATE(L) are nonetheless linkable on the given routing.

Given these routing and assignment strategies, and an estimate of cell yield, we would like to know what values for K and L are necessary as well as what physical wafer dimensions are best in order to guarantee good wafer yield. A simple analytical expression can give the expected wafer yield assuming no contraints on allowable assignments. This formula can be nested to handle the case of unrestricted SKIP without allowing MIGRATION. More realistic cases of limited SKIP with MIGRATION are too difficult to handle analytically and require Monte Carlo simulation. An assignment program for the SKIP(K) MIGRATE(L) strategy is being written at present. The program will be run repetitively on simulated defective wafers to gather yield statistics. A small nonconstructive simulation program has been written which can handle the SKIP(K) MIGRATE(1) case.

As the 200-MHz input rate is too fast for current MOS technology, a separate bipolar 8-bit serial-to-parallel converter will be used to drive eight parallel count-enable registers. Figure V-6 shows the partitioning of the integrator into eight 32-counter columns. The duration of the 256-bit window is 1.28 μs and the windows repeat with a 10.24-μs period. Thus there are more than 8 μs for the count, parallel transfer, and preset actions to take place. The stressing readout case is a short integration period; with a 10-frame integration, the readout transfer must be at 2.5 MW/s. The only high-speed circuitry is in the count-enable register.

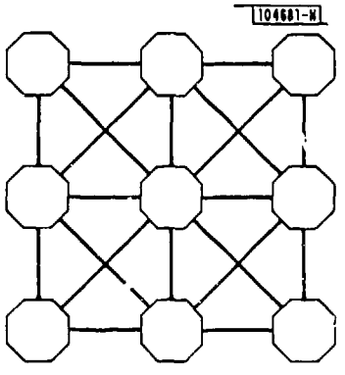


Fig. V-1. A systolic array of octagonal cells.

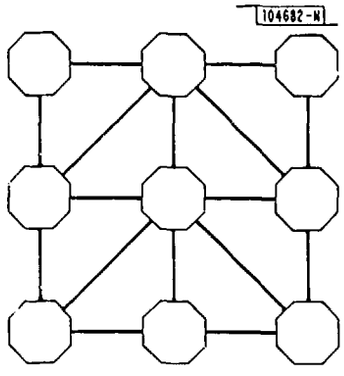


Fig. V-2. Hexagonal-cell array formed from the octagonal-cell array.

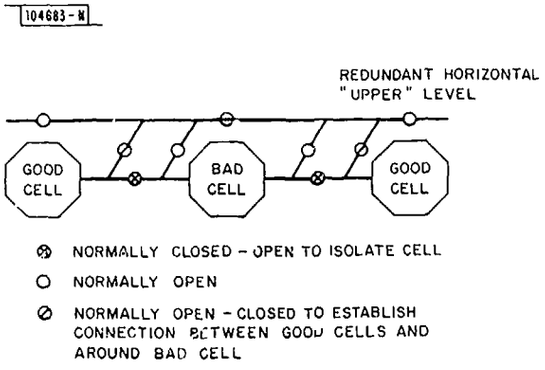


Fig. V-3. Routing of a signal around a defective cell.

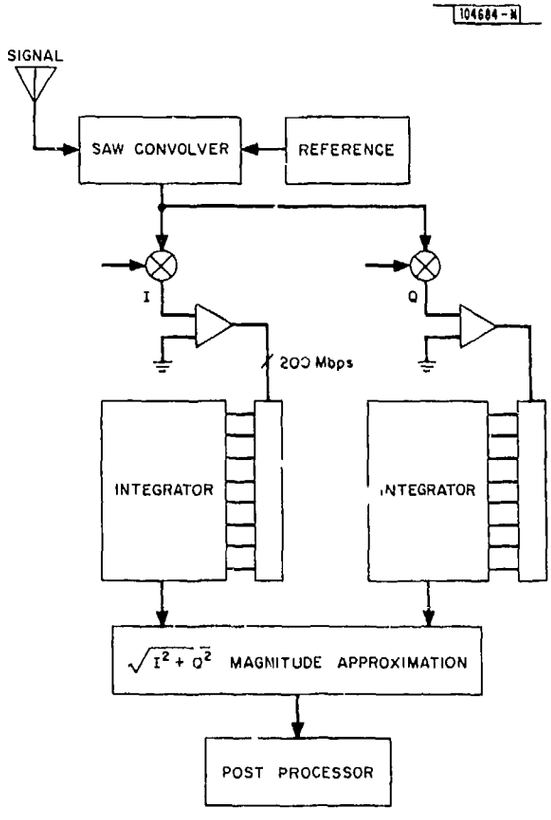


Fig. V-4. Hybrid integrating correlator.

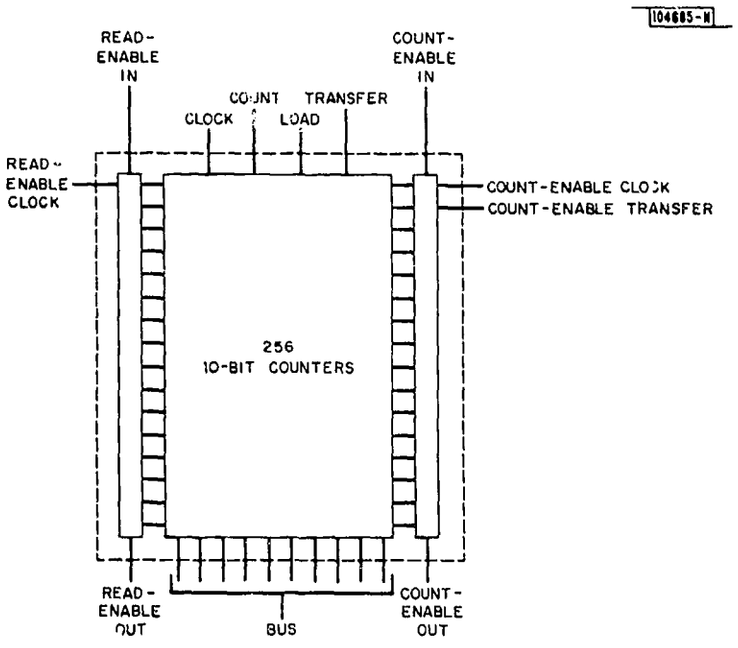


Fig. V-5. Functional block diagram of integrator.

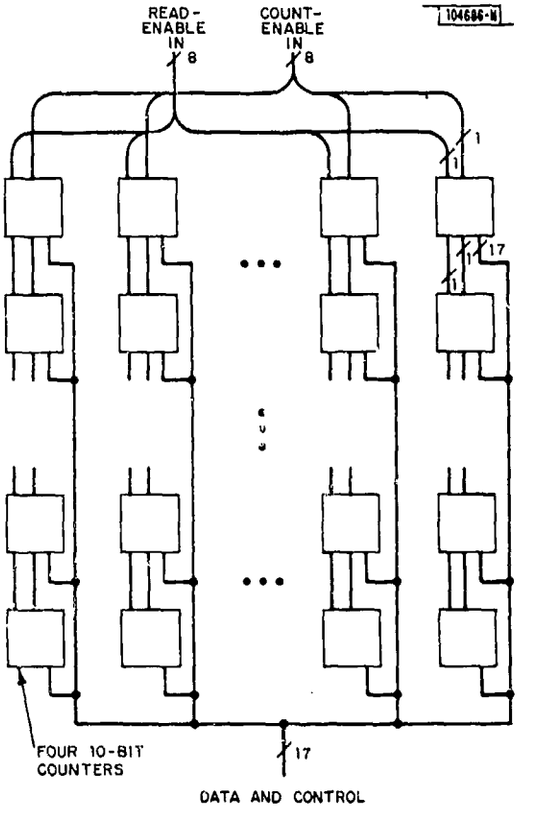


Fig. V-6. Block diagram of partitioned integrator.

REFERENCES

1. J. I. Raffel, "On the Use of Nonvolatile Programmable Links for Restructurable VLSI," Proceedings of Cal Tech Conference on Very Large Scale Integration, January 1979, pp. 95-104.

2. A. Kondo et al., "Dynamic Injection MNOS Memory Devices," Jpn. J. Appl. Phys. 19 (1980), Supp. 19-1, pp. 231-237.

3. M. R. Barbacci, "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Trans. Computers C-24, No. 2, 137-150 (February 1975).

4. S. A. Shiva, "Computer Hardware Description Languages - A Tutorial," Proc. IEEE 67, No. 12, 1605-1615 (December 1979).

5. F. J. Hill and Z. Navabi, "Extending Second Generation AHPL Software to Accommodate AHPL III," Proceedings of the 4th International Symposium on Computer Hardware Description Languages, Palo Alto, California, October 1979, pp. 47-53.

6. F. J. Hill and A. R. Peterson, Digital Systems: Hardware Organization and Design, Second Edition (Wiley, New York, 1978).

7. G. F. Maxey and E. I. Organick, "CASL - A Language for Automating the Implementation of Computer Architectures," Proceedings of the 4th International Symposium on Computer Hardware Description Languages, Palo Alto, California, October 1979, pp. 102-108.

8. W. M. Van Cleemput, "A Hierarchical Language for the Structural Description of Digital Systems," Proceedings of the 14th Design Automation Conference, June 1977, pp. 377-385.

9. T. M. McWilliams, L. C. Widdoes, Jr., and L. L. Wood, "Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project," Report UCID-17705, Lawrence Livermore Laboratory (1977).

10. T. M. McWilliams and L. C. Widdoes, Jr., "SCALD: Structured Computer-Aided Logic Design," Proceedings of the 15th Design Automation Conference, June 1978, pp. 271-277.

11. T. M. McWilliams and L. C. Widdoes, Jr., "The SCALD Physical Design Subsystem," Proceedings of the 15th Design Automation Conference, June 1978, pp. 278-284.

12. L. E. Heindel and J. T. Roberts, LANG-PAK - An Interactive Language Design System (Elsevier North-Holland, New York, 1975).

13. R. Piloty et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Basic Principles," to be published in the Proceedings of the 1980 National Computer Conference, Vol. 49.

14. R. Piloty et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Derivation," to be published in the Proceedings of the 1980 National Computer Conference, Vol. 49.

15. R. Piloty et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Application," to be published in the Proceedings of the 1980 National Computer Conference, Vol. 49.

16. H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," in Introduction to VLSI Systems, by C. Mead and L. Conway (Addison-Wesley, Reading, Massachusetts, 1980), Sec. 8.3, pp. 271-292.

APPENDIX
SYNTAX OF HIERARCHICAL AND ITERATIVE STRUCTURE DESCRIPTION LANGUAGE

The HISDL syntax description uses the following notation:

(1) The equal sign "=" is to be read as "is defined as".

(2) <character>, <letter>, <digit>, and <integer> represent the sets of characters, letters, digits, and integers (signed), respectively.

(3) A literal is underlined, e.g., STRUCTURE, =, |.

(4) The vertical bar "|" is used to separate alternatives in the definitions.

(5) Braces, for example in $\{\text{<character>}\}_0^{79}$, specify the number of repetition that is allowed. The lower limit is always specified and it can be zero. The upper limit when not specified means that there is no set upper limit on the number of repetitions. However, for any physical implementation of the language, there is always an upper limit be it disk space for the HISDL source file, memory space or word size limitation. Also the unspecified upper limit may be known only after the data base has been implemented.

SYNTAX OF HISDL VERSION 1.0

<program> = $\{\text{<comment>}|\text{<type definition>}\}_1$

<comment> = % $\{\text{<character>}\}_0^{79}$

<type definition> = (<sheader> <body> <stail>|<cheader> <body> <ctail>)

<sheader> = STRUCTURE <header>

<header> = <type name> ($\{\text{<parameter list>}\}_0^1$)

<cheader> = CELL <header>

<type name> = <name>

<name> = <letter> $\{\text{<letter>}|\text{<digit>}\}_0^7$

<parameter list> = <parameter> $\{$, <parameter>$\}_0^{100}$

<parameter> = <identifier type>

<identifier type> = <array name type>|<name>

<array name type> = <name> <left bracket> <index specification list> <right bracket>

<left bracket> = { | for EBCDIC character set
                 { [for ASCII character set

<right bracket> = { | for EBCDIC character set
                  {] for ASCII character set

<index specification list> = <index specification> $\{$, <index specification>$\}_0^3$

<index specification> = <limit1> $\{$:<limit2>$\}_0^1$ $\{$;<increment>$\}_0^1$

<limit1> = <integer>

<limit2> = <integer>

<increment> = <integer>

<body> = <io declaration>|<component declaration>|<type definition>
|<connection body>

<io declaration> = <io type> <io list>

<io type> = IN|OUT|INOUT

<io list> = <io name type> $\{$, <io name type>$\}_0^{100}$

<io name type> = <identifier type> $\{$<path width specification>$\}_0^1$

<path width specification> = < <index specification> >

<component declaration> = COMPONENTS <component declaration list>

<component declaration list> = <component list> $\{$, <component list>$\}_0^{100}$

<component list> = <name type list> : <type name>

<name type list> = <identifier type> $\{$, <identifier type>$\}_0^{100}$

<connection body> = BEGIN $\{$<connection statement>$\}_0$ END

<connection statement> = <connection list>|<component invocation>|<for statement>

<connection list> = $\{$<net name>$\}_0^1$ <left brace> <pin list> <right brace>

<net name> = <identifier>

<identifier> = <array name>|<name>

<array name> = <name> <left bracket> <index list> <right bracket>

<index list> = <index range> $\{$, <index range>$\}_0^3$

<index range> = <limit1> $\{$: <limit2>$\}_0^1$

left brace = { / for EBCDIC character set
{ { for ASCII character set

right brace = { / for EBCDIC character set
{ } for ASCII character set

<pin list> = <pin name> $\{$, <pin name>$\}_0^{100}$

<pin name> = <component name> . <io name>

<component name> = <identifier>

<io name> = <identifier> $\{$<path width>$\}_0^1$

<path width> = < <index range> >

<component invocation> = <component name> ({<pin list>}$_0^1$)

<for statement> = <for header> <for body> <for tail>

<for header> = FOR <control variable> = <limit1> TO <limit2> {BY <increment>}$_0^1$

<control variable> = <name>

<for body> = <for connection list>|<for component invocation>|<for body>

<for connection list> = <for name> <left brace> <for pin list> <right brace>

<for name> = <for array name>|<name>

<for array name> = <name> <left bracket> <for index list> <right bracket>

<for index list> = <for index range> {, <for index range>}$_0^3$

<for index range> = <for limit1> {: <for limit2>}$_0^1$

<for limit1> = <for expression>

<for limit2> = <for expression>

<for expression> = <integer>|<for term> {<op> <for term>}$_0^{10}$

<for term> = {<sign>}$_0^1$ (<control variable>|<integer>)

<sign> = + | -

<op> = + | - | * | /

<for pin list> = <for pin name> {, <for pin name>}$_0^{100}$

<for pin name> = <for component name> . <for io name>

<for component name> = <for name>

<for io name> = <for name> {<for path width>}$_0^1$

<for path width> = < <for index range> >

<for component invocation> = <for component name> ({<for pin list>}$_0^1$)

<for tail> = ENDFOR

<stail> = ENDSTRUCT {<type name>}$_0^1$

<ctail> = ENDCELL {<type name>}$_0^1$

GLOSSARY

CMOS	Complementary Metal-Oxide Semiconductor
FAMOS	Floating-Gate Avalanche-Injection MOS
FF	Flip Flop
FFT	Fast Fourier Transform
HISDL	Hierarchical and Iterative Structure Description Language
LSI	Large Scale Integration
MNOS	Metal-Nitride-Oxide Semiconductor
MOS	Metal-Oxide Semiconductor
MSI	Medium Scale Integration
PLA	Programmable Logic Array
PROM	Programmable Read-Only Memory
RVLSI	Restructurable Very Large Scale Integration
SAW	Surface Acoustic Wave
SIMD	Single-Instruction Multiple Data
VLSI	Very Large Scale Integration

UNCLASSIFIED (16) 0D30

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(19) **REPORT DOCUMENTATION PAGE**

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (18) ESD-TR-80-192	2. GOVT ACCESSION NO. AD-A096075
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)* (6) Restructurable VLSI Program •	5. TYPE OF REPORT & PERIOD COVERED (9) Semiannual Technical Summary rept. 1 Apr 1979 – 31 Mar 1980
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Allan H. Anderson	8. CONTRACT OR GRANT NUMBER(s) (15) F19628-80-C-0002 ARPA Order-3797
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 3797 Program Element No. 61101E Project No. 0D30
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE (11) 31 Mar 1980
	13. NUMBER OF PAGES 36
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Electronic Systems Division Hanscom AFB Bedford, MA 01731 (12) 34	15. SECURITY CLASS. *(of this report)* Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

VLSI
Restructurable VLSI (RVLSI)
programmable interconnect
defect avoidance
customization
hardware description language
placement
routing
systolic array
integrator

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This initial report describes work on the Restructurable VLSI Research Program sponsored by the Information Processing Techniques Office of the Defense Advanced Research Projects Agency during the two semiannual periods, covering 1 April 1979 through 31 March 1980.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*